Universidad San Sebastián
Facultad de Ingeniería y Tecnología
ICIF H001 – Inteligencia Artificial

Solemne 3

Reto para Búsqueda de Caminos Bi-Objetivos

Inteligencia Artificial

Integrantes: Felipe Correa González
Jairo González Urbina
Lenny Roa Sepúlveda

Profesor: Carlos Hernández Ulloa

NRC Teo: 6249

10 de diciembre de 2024

Índice

1. Introducción

En el siguiente informe, presentaremos la solución al reto propuesto, el problema consistía en encontrar una propuesta de Pareto en un grafo dirigido que conecta el estado inicial con un estado de destino final pasando por múltiples destinos entremedios.

Nuestro objetivo era implementar una solución eficiente que permitiera resolver instancias del problema en los mapas optimizando tiempo de ejecución y número de nodos generados expandidos.

El actual informe, describe el desarrollo del trabajo, incluyendo las modificaciones realizadas al código inicial y los resultados obtenidos.

2. Metodología

Análisis del problema:

El reto planteado por el profesor consistía en resolver el BCBiMDS utilizando una versión eficiente del algoritmo BOA*. Los datos se proporcionaron en formato de texto, incluyendo mapas y secuencias de destinos.

Nuestro enfoque se basó en modificar y optimizar el código original para mejorar la eficiencia en la búsqueda de soluciones y adaptarlo a nuestras necesidades.

Modificaciones realizadas:

El código inicial estaba en C y usaba el mapa de Nueva York “NY-road-d” con la secuencia de destinos” NY-queries-2p”.

1. Renombramos este código a “RetoIA” y realizamos ajustes para integrarlo con scripts adicionales en Python.

2. Agregamos nuevas funciones para analizar. Optimizar la lista de nodos y paradas. Además de que guardara todos los nodos que utiliza al leer el mapa para utilizarlos en un nuevo mapa

3. Creación de “NodosUtilizados.txt”: El cual genera una lista de puntos intermedios entre las paradas para facilitar la creación del nuevo mapa.

Implementación en Python:

El desarrollo se llevó a cabo en las siguientes etapas:

1. En el “ListaParadas” La llamada de “RetoIA.exe” con las instancias y segmentos correspondientes de “NY-queries-2p.txt” da el “output.txt”.
2. Generación de “output.txt”: Vemos que este contiene las soluciones obtenidas en cada ejecución en el mapa original.
3. En el “ListaParadasFinal” La llamada de “RetoIA - Final.exe” con las instancias y segmentos correspondientes de “NY-queries-2p.txt” da el “output2.txt”.
4. Generación de “output.txt”: Vemos que este contiene las soluciones obtenidas en cada ejecución en el mapa original.

Optimización del mapa de nodos:

Usamos el script “relacionarTXT.py” para procesar el mapa de Nueva York y combinarlo con los nodos utilizados, generando un mapa optimizado de nodos.

Cálculo final:

Un script llamado “ListaParadasFinal.py” analiza los segmentos entre cada parada y genera nuevas soluciones basadas en el mapa optimizado el filtered_NY-road-d.txt.

Ejecución principal:

Tenemos que el “ScriptIA.py” coordina todos los procesos y genera los resultados finales.

3. Resultados

Salida generada por código

El proceso produce varios archivos de salida:

“output2.txt”: Lista de todas las soluciones encontradas, incluyendo detalles de tiempo de ejecución y nodos utilizados.

“NodosUtilizados.txt” y “NodosUtilizados2.txt” : Representación optimizada del mapa de nodos relevantes.

“filtered_NY-road-d.txt”: mapa final de las paradas procesadas.

Ejemplo de una solución generada:

Processing segment: 2995 -> 2515

Segment results: Runtime 4.00 ms, Solutions 0, Expansions 0, Generated 0

Processing segment: 2515 -> 1443

Segment results: Runtime 2.00 ms, Solutions 0, Expansions 6, Generated 5

Processing segment: 1443 -> 20386

Segment results: Runtime 31.00 ms, Solutions 0, Expansions 1, Generated 5

Optimización lograda:

Vemos una reducción significativa en los nodos generados al usar el mapa optimizado, ya que tenemos una mayor eficiencia al procesar múltiples instancias de destinos.

El uso de Python nos permitió la implementación de los scripts y facilitó la manipulación de los archivos generados.

Sin embargo, se identificaron algunos posibles desafíos futuros:

- Tenemos que el procesamiento de grandes cantidades de datos en C puede ser difícil ya que no pudimos hacer todo en C porque se llenaba la memoria.
- Tuvimos dificultades iniciales en la integración de los scripts.

Pero a pesar de estas limitaciones, se logró optimizar significativamente el proceso y cumplir con los requisitos del reto.

4. Conclusión

El trabajo desarrollado nos permitió resolver instancias del problema BCBiMDS de manera eficiente. Tuvimos que las modificaciones realizadas al código original y la implementación de nuevos scripts en Python mejoraron el rendimiento y optimizaron los resultados obtenidos. Sin duda, este proyecto representa un avance en el desarrollo de algoritmos para problemas de optimización en grafos.

5. Anexo Pc

El pc que se usó para las pruebas es:

- Procesador: AMD Ryzen 5 7600X 6-Core Processor 4.70GHz

Ram: 32GB DDR5

6. Anexo Código

https://github.com/Feltrix9/IA_proyecto

RetoIA.c :

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <sys/time.h> // Added to define struct timeval

// #include <unistd.h> // Removed as it is not necessary for this code


#define MAXNODES 4000000

#define MAXNEIGH 45

#define MAX_SOLUTIONS 1000000

#define MAX_RECYCLE   100000


#define LARGE  1000000000

#define BASE   10000000


#define max(x,y) ( (x) > (y) ? (x) : (y) )

#define min(x,y) ( (x) < (y) ? (x) : (y) )


//********************************************** Main data structures ****************************************************

struct gnode;
```

```
typedef struct gnode gnode;

typedef struct {
   int id;       // ID del nodo
   int coordX;    // Coordenada X
   int coordY;    // Coordenada Y
} Nodo;

struct gnode // stores info needed for each graph node
{
 long long int id;
 unsigned h1;
 unsigned h2;
 unsigned long long int key;
 unsigned gmin;
 unsigned long heapindex;
};

struct snode;
typedef struct snode snode;

struct snode // BOA*'s search nodes
{
 int cost; // Costo acumulado
 int stops; // Número de paradas
 struct snode* prev; // Nodo anterior en la ruta
 // Otros campos necesarios
 int state;
 unsigned g1;
 unsigned g2;
 double key;
```

```
 unsigned long heapindex;
 snode *searchtree;
};


// Variables globales
Nodo *nodosUtilizados = NULL;  // Arreglo din□mico para guardar nodos utilizados
int totalNodosUtilizados = 0; // Contador de nodos utilizados

gnode* graph_node;
unsigned num_gnodes;
unsigned adjacent_table[MAXNODES][MAXNEIGH];
unsigned pred_adjacent_table[MAXNODES][MAXNEIGH];
unsigned goal, start;
gnode* start_state;
gnode* goal_state;
snode* start_node;

unsigned long long int stat_expansions = 0;
unsigned long long int stat_generated = 0;
unsigned long long int minf_solution = LARGE;

unsigned solutions[MAX_SOLUTIONS][2];
unsigned nsolutions = 0;
unsigned stat_pruned = 0;
unsigned stat_created = 0;


//********************************************** Binary Heap Data Structures
***************************************************
```

```
// -------------------------   Binary Heap for Dijkstra  ---------------------------------------
#define HEAPSIZEDIJ 3000000
gnode* heap_dij[HEAPSIZEDIJ];
unsigned long int heapsize_dij = 0;
unsigned long int stat_percolations = 0;

// ------------------------------------------------------------
void percolatedown_dij(int hole, gnode* tmp) {
 int child;

 if (heapsize_dij != 0) {
  for (; 2 * hole <= heapsize_dij; hole = child) {
   child = 2 * hole;
   if (child != heapsize_dij && heap_dij[child + 1]->key < heap_dij[child]->key)
    ++child;
   if (heap_dij[child]->key < tmp->key) {
    heap_dij[hole] = heap_dij[child];
    heap_dij[hole]->heapindex = hole;
    ++stat_percolations;
   }
   else
    break;
  } // end for
  heap_dij[hole] = tmp;
  heap_dij[hole]->heapindex = hole;
 }
}
/* ------------------------------------------------------------ */
void percolateup_dij(int hole, gnode* tmp) {
 if (heapsize_dij != 0) {
```

```
    for (; hole > 1 && tmp->key < heap_dij[hole / 2]->key; hole /= 2) {
     heap_dij[hole] = heap_dij[hole / 2];
     heap_dij[hole]->heapindex = hole;
     ++stat_percolations;
    }
    heap_dij[hole] = tmp;
    heap_dij[hole]->heapindex = hole;
  }
}
/* ------------------------------------------------------------- */
void percolateupordown_dij(int hole, gnode* tmp) {
  if (heapsize_dij != 0) {
    if (hole > 1 && heap_dij[hole / 2]->key > tmp->key)
      percolateup_dij(hole, tmp);
    else
      percolatedown_dij(hole, tmp);
  }
}
/* ------------------------------------------------------------- */
void insertheap_dij(gnode* thiscell) {

  if (thiscell->heapindex == 0)
    percolateup_dij(++heapsize_dij, thiscell);
  else
    percolateupordown_dij(thiscell->heapindex, heap_dij[thiscell->heapindex]);
}
/* ------------------------------------------------------------- */
void deleteheap_dij(gnode* thiscell) {
  if (thiscell->heapindex != 0) {
    percolateupordown_dij(thiscell->heapindex, heap_dij[heapsize_dij--]);
    thiscell->heapindex = 0;
```

```
  }
}
/* ------------------------------------------------------------ */
gnode* topheap_dij() {
 if (heapsize_dij == 0)
   return NULL;
 return heap_dij[1];
}
/* ------------------------------------------------------------ */
void emptyheap_dij() {
 int i;

 for (i = 1; i <= heapsize_dij; ++i)
   heap_dij[i]->heapindex = 0;
 heapsize_dij = 0;
}

/* ------------------------------------------------------------ */
gnode* popheap_dij() {
 gnode* thiscell;

 if (heapsize_dij == 0)
   return NULL;
 thiscell = heap_dij[1];
 thiscell->heapindex = 0;
 percolatedown_dij(1, heap_dij[heapsize_dij--]);
 return thiscell;
}

int sizeheap_dij() {
 return heapsize_dij;
```

```
}

gnode* posheap_dij(int i) {
 return heap_dij[i];
}

// -------------------------   Binary Heap for BOA*  ---------------------------------------
#define HEAPSIZE 40000000
snode* heap[HEAPSIZE];
unsigned long int heapsize = 0;

// ------------------------------------------------------------
void percolatedown(int hole, snode* tmp) {
 int child;

 if (heapsize != 0) {
  for (; 2 * hole <= heapsize; hole = child) {
   child = 2 * hole;
   if (child != heapsize && heap[child + 1]->key < heap[child]->key)
    ++child;
   if (heap[child]->key < tmp->key) {
    heap[hole] = heap[child];
    heap[hole]->heapindex = hole;
    ++stat_percolations;
   }
   else
    break;
  } // end for
  heap[hole] = tmp;
  heap[hole]->heapindex = hole;
 }
```

```
}
/* ------------------------------------------------------------ */
void percolateup(int hole, snode* tmp) {
 if (heapsize != 0) {
  for (; hole > 1 && tmp->key < heap[hole / 2]->key; hole /= 2) {
   heap[hole] = heap[hole / 2];
   heap[hole]->heapindex = hole;
   ++stat_percolations;
  }
  heap[hole] = tmp;
  heap[hole]->heapindex = hole;
 }
}
/* ------------------------------------------------------------ */
void percolateupordown(int hole, snode* tmp) {
 if (heapsize != 0) {
  if (hole > 1 && heap[hole / 2]->key > tmp->key)
   percolateup(hole, tmp);
  else
   percolatedown(hole, tmp);
 }
}
/* ------------------------------------------------------------ */
void insertheap(snode* thiscell) {
 if (thiscell->heapindex == 0)
  percolateup(++heapsize, thiscell);
 else
  percolateupordown(thiscell->heapindex, heap[thiscell->heapindex]);
}
/* ------------------------------------------------------------ */
void deleteheap(snode* thiscell) {
```

```
  if (thiscell->heapindex != 0) {
    percolateupordown(thiscell->heapindex, heap[heapsize--]);
    thiscell->heapindex = 0;
  }
}
/* ------------------------------------------------------------- */
snode* topheap() {
  if (heapsize == 0)
    return NULL;
  return heap[1];
}
/* ------------------------------------------------------------- */
void emptyheap() {
  int i;

  for (i = 1; i <= heapsize; ++i)
    heap[i]->heapindex = 0;
  heapsize = 0;
}

/* ------------------------------------------------------------- */
snode* popheap() {
  snode* thiscell;

  if (heapsize == 0)
    return NULL;
  thiscell = heap[1];
  thiscell->heapindex = 0;
  percolatedown(1, heap[heapsize--]);
  return thiscell;
}
```

```
int sizeheap() {
 return heapsize;
}

long int opensize() {
 return heapsize_dij;
}
snode* posheap(int i) {
 return heap[i];
}
// -------------------------   Binary Heap end ----------------------------------------
```

```
//*********************************************  Reading  the  file
*****************************************************

void read_adjacent_table(const char* filename) {
        FILE* f;
        int i, ori, dest, dist, t;
        f = fopen(filename, "r");
        int num_arcs = 0;
        if (f == NULL)    {
                printf("Cannot open file %s.\n", filename);
                exit(1);
        }
        fscanf(f, "%d %d", &num_gnodes, &num_arcs);
        fscanf(f, "\n");
//      printf("%d %d", num_gnodes, num_arcs);
```

```
        for (i = 0; i < num_gnodes; i++)
                adjacent_table[i][0] = 0;


        for (i = 0; i < num_arcs; i++) {
                fscanf(f, "%d %d %d %d\n", &ori, &dest, &dist, &t);
        //      printf("%d %d %d %d\n", ori, dest, dist, t);
                adjacent_table[ori - 1][0]++;
                adjacent_table[ori - 1][adjacent_table[ori - 1][0] * 3 - 2] = dest - 1;
                adjacent_table[ori - 1][adjacent_table[ori - 1][0] * 3 - 1] = dist;
                adjacent_table[ori - 1][adjacent_table[ori - 1][0] * 3] = t;


                pred_adjacent_table[dest - 1][0]++;
                pred_adjacent_table[dest - 1][pred_adjacent_table[dest - 1][0] * 3 - 2] = ori
- 1;
                pred_adjacent_table[dest - 1][pred_adjacent_table[dest - 1][0] * 3 - 1] = dist;
                pred_adjacent_table[dest - 1][pred_adjacent_table[dest - 1][0] * 3] = t;
        }
        fclose(f);
}


void new_graph() {
        int y;
        if (graph_node == NULL) {
                graph_node = (gnode*) calloc(num_gnodes, sizeof(gnode));
                for (y = 0; y < num_gnodes; ++y)                {
                        graph_node[y].id = y;
                        graph_node[y].gmin = LARGE;
                        graph_node[y].h1 = LARGE;
                        graph_node[y].h2 = LARGE;
                }
        }
```

```
}


//*********************************************                                    BOA*
*****************************************************


void initialize_parameters() {
  start_state = &graph_node[start];
  goal_state = &graph_node[goal];
  stat_percolations = 0;
}

int backward_dijkstra(int dim) {
  int i;
        for (i = 0; i < num_gnodes; ++i)
    graph_node[i].key = LARGE;
  emptyheap_dij();
  goal_state->key = 0;
  insertheap_dij(goal_state);

  while (topheap_dij() != NULL) {
    gnode* n;
    gnode* pred;
    short d;
    n = popheap_dij();
    if (dim == 1)
      n->h1 = n->key;
    else
      n->h2 = n->key;
    ++stat_expansions;
    for (d = 1; d < pred_adjacent_table[n->id][0] * 3; d += 3) {
```

```
            pred = &graph_node[pred_adjacent_table[n->id][d]];
            int new_weight = n->key + pred_adjacent_table[n->id][d + dim];
            if (pred->key > new_weight) {
              pred->key = new_weight;
              insertheap_dij(pred);
            }
          }
        }
        return 1;
    }

    snode* new_node() {
        snode* state = (snode*)malloc(sizeof(snode));
        state->heapindex = 0;
        return state;
    }

    int boastar() {
        FILE* f = fopen("Txt/NodosUtilizados.txt", "a");
        if (f == NULL) {
            perror("Error al abrir el archivo para guardar los nodos utilizados");
            exit(1);
        }

        snode* recycled_nodes[MAX_RECYCLE];
        int next_recycled = 0;
        nsolutions = 0;
        stat_pruned = 0;
        emptyheap();

        start_node = new_node();
```

```
    ++stat_created;
    start_node->state = start;
    start_node->g1 = 0;
    start_node->g2 = 0;
    start_node->key = 0;
    start_node->searchtree = NULL;
    insertheap(start_node);

    stat_expansions = 0;
    while (topheap() != NULL) {
      snode* n = popheap(); // Best node in open
      short d;

      if (n->g2 >= graph_node[n->state].gmin || n->g2 + graph_node[n->state].h2 >= minf_solution) {
        stat_pruned++;
        if (next_recycled < MAX_RECYCLE) {
          recycled_nodes[next_recycled++] = n;
        }
        continue;
      }

      graph_node[n->state].gmin = n->g2;

      // Guardar el nodo expandido
      fprintf(f, "Nodo expandido: %d (g1: %d, g2: %d)\n", n->state, n->g1, n->g2);

      if (n->state == goal) {
        printf("GOAL [%d,%d] nsolutions:%d expanded:%llu generated:%llu heapsize:%d pruned:%d\n",
            n->g1, n->g2, nsolutions, stat_expansions, stat_generated, sizeheap(), stat_pruned);
```

```
      solutions[nsolutions][0] = n->g1;
      solutions[nsolutions][1] = n->g2;
      nsolutions++;
      if (nsolutions > MAX_SOLUTIONS) {
        printf("Maximum number of solutions reached, increase MAX_SOLUTIONS!\n");
        fclose(f);
        exit(1);
      }
      if (minf_solution > n->g2)
        minf_solution = n->g2;
      continue;
    }

    ++stat_expansions;

    for (d = 1; d < adjacent_table[n->state][0] * 3; d += 3) {
      snode* succ;
      double newk1, newk2, newkey;
      unsigned nsucc = adjacent_table[n->state][d];
      unsigned cost1 = adjacent_table[n->state][d + 1];
      unsigned cost2 = adjacent_table[n->state][d + 2];

      unsigned newg1 = n->g1 + cost1;
      unsigned newg2 = n->g2 + cost2;
      unsigned h1 = graph_node[nsucc].h1;
      unsigned h2 = graph_node[nsucc].h2;

      if (newg2 >= graph_node[nsucc].gmin || newg2 + h2 >= minf_solution)
        continue;

      newk1 = newg1 + h1;
```

```
        newk2 = newg2 + h2;

        if (next_recycled > 0) { // To reuse pruned nodes in memory
          succ = recycled_nodes[--next_recycled];
        } else {
          succ = new_node();
          ++stat_created;
        }

        succ->state = nsucc;
        stat_generated++;

        newkey = newk1 * (double)BASE + newk2;
        succ->searchtree = n;
        succ->g1 = newg1;
        succ->g2 = newg2;
        succ->key = newkey;
        insertheap(succ);
      }
    }

    fclose(f);
    return nsolutions > 0;
}



/* -----------------------------------------------------------------------*/
void call_boastar() {
    float runtime;
```

```
    struct timeval tstart, tend;
    unsigned long long min_cost;
    unsigned long long min_time;

    initialize_parameters();

    gettimeofday(&tstart, NULL);

    //Dijkstra h1
    if (backward_dijkstra(1))
      min_cost = start_state->h1;

    //Dijkstra h2
    if (backward_dijkstra(2))
      min_time = start_state->h2;


        //BOA*
    boastar();


    gettimeofday(&tend, NULL);
    runtime = 1.0 * (tend.tv_sec - tstart.tv_sec) + 1.0 * (tend.tv_usec - tstart.tv_usec) / 1000000.0;
    // printf("nsolutions:%d Runtime(ms):%f Generated: %llu statexpanded1:%llu\n", nsolutions, time_astar_first1*1000, stat_generated, stat_expansions);
    printf("%lld;%lld;%d;%f;%llu;%llu;%lu;%llu\n",
      start_state->id + 1,
      goal_state->id + 1,
      nsolutions,
      runtime * 1000,
```

```
        stat_generated,
        stat_expansions,
        stat_created,
        stat_percolations);
}

void execute_with_stops(const int stops[], int num_stops) {
  unsigned long long total_expansions = 0, total_generated = 0;
  float total_runtime = 0;
  unsigned total_solutions = 0;

  struct timeval tstart, tend;
  int i;

  for (i = 0; i < num_stops - 1; i++) {
    start = stops[i];     // Nodo inicial del segmento
    goal = stops[i + 1];  // Nodo final del segmento
    printf("Processing segment: %d -> %d\n", start, goal);

    start_state = &graph_node[start];
    goal_state = &graph_node[goal];

    gettimeofday(&tstart, NULL);

    // Inicializar par�metros y ejecutar BOA* para este segmento
    initialize_parameters();
    if (backward_dijkstra(1) && backward_dijkstra(2)) {
      boastar();
    }

    gettimeofday(&tend, NULL);
```

```
    // Calcular tiempo de ejecuci�n para este segmento
    float runtime = 1.0 * (tend.tv_sec - tstart.tv_sec) +
            1.0 * (tend.tv_usec - tstart.tv_usec) / 1000000.0;



    // Acumular m�tricas globales
    total_runtime += runtime;
    total_expansions += stat_expansions;
    total_generated += stat_generated;
    total_solutions += nsolutions;

    // Mostrar resultados del segmento
    printf("Segment results: Runtime %.2f ms, Solutions %d, Expansions %llu, Generated
%llu\n",
        runtime * 1000, nsolutions, stat_expansions, stat_generated);


  }


  // Mostrar resultados consolidados
  printf("Total Results:\n");
  printf("Runtime: %.2f ms\n", total_runtime * 1000);
  printf("Total Expansions: %llu\n", total_expansions);
  printf("Total Generated: %llu\n", total_generated);
  printf("Total Solutions: %u\n", total_solutions);
}

/*-------------------------------------------------------------------------*/

// Implementaci�n para guardar nodos utilizados
void guardarNodosUtilizados(const char *nombreArchivo) {
```

```
    FILE *archivo = fopen(nombreArchivo, "a");
    if (archivo == NULL) {
        perror("Error al abrir el archivo de salida");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < totalNodosUtilizados; i++) {
        fprintf(archivo, "%d %d %d\n", nodosUtilizados[i].id, nodosUtilizados[i].coordX, nodosUtilizados[i].coordY);
    }

    fclose(archivo);
    printf("Nodos utilizados guardados en %s\n", nombreArchivo);
}

// Simulaci�n de la funci�n que copia un nodo
void copiarNodo(int idNodo, Nodo *destino) {
    // Aqu� se llenar�a el nodo `destino` con los datos reales desde el grafo.
    // Simulamos con datos ficticios:
    destino->id = idNodo;
    destino->coordX = idNodo * 10;  // Supongamos coordenada X
    destino->coordY = idNodo * 20;  // Supongamos coordenada Y
}

// Agregar un nodo a la lista de nodos utilizados
void agregarNodoUtilizado(int idNodo) {
    if (totalNodosUtilizados >= 1000) {
        // Redimensionar din�micamente si es necesario
        nodosUtilizados = realloc(nodosUtilizados, (totalNodosUtilizados + 500) * sizeof(Nodo));
        if (nodosUtilizados == NULL) {
```

```
            perror("Error al redimensionar el array de nodos utilizados");
            exit(EXIT_FAILURE);
        }
    }

    copiarNodo(idNodo, &nodosUtilizados[totalNodosUtilizados]);
    totalNodosUtilizados++;
}

// Implementaci�n simulada de `execute_with_stops`



int main(int argc, char *argv[]) {
    // Verificar que se han pasado suficientes argumentos
    if (argc < 2) {
        printf("Se necesitan al menos un argumento para las paradas.\n");
        return 1;  // Salir si no se pasa ning�n argumento
    }

    // Convertir los argumentos a enteros y guardarlos en un array
    int num_stops = argc - 1;
    int stops[num_stops];

    for (int i = 1; i < argc; i++) {
        stops[i - 1] = atoi(argv[i]);
    }

    // Leer el grafo desde el archivo
    read_adjacent_table("Txt/NY-road-d.txt");
    new_graph();
```

```
    // Reservar memoria inicial para nodos utilizados
    nodosUtilizados = malloc(1000 * sizeof(Nodo));  // Tama◆o inicial arbitrario
    if (nodosUtilizados == NULL) {
        perror("Error al reservar memoria para nodos utilizados");
        return 1;
    }

    // Ejecutar BOA* para cada segmento entre paradas
    execute_with_stops(stops, num_stops);

    // Guardar nodos utilizados en un archivo
    guardarNodosUtilizados("Txt/NodosUtilizados.txt");

    // Liberar memoria
    free(nodosUtilizados);

    return 0;
}
```

RetoIA - Final.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h> // Added to define struct timeval
// #include <unistd.h> // Removed as it is not necessary for this code

#define MAXNODES 4000000
#define MAXNEIGH 45
#define MAX_SOLUTIONS 1000000
#define MAX_RECYCLE   100000
```

```
#define LARGE  1000000000
#define BASE   10000000

#define max(x,y) ( (x) > (y) ? (x) : (y) )
#define min(x,y) ( (x) < (y) ? (x) : (y) )

//*********************************************** Main data structures ****************************************************

struct gnode;
typedef struct gnode gnode;

typedef struct {
   int id;       // ID del nodo
   int coordX;    // Coordenada X
   int coordY;    // Coordenada Y
} Nodo;

struct gnode // stores info needed for each graph node
{
 long long int id;
 unsigned h1;
 unsigned h2;
 unsigned long long int key;
 unsigned gmin;
 unsigned long heapindex;
};

struct snode;
typedef struct snode snode;

struct snode // BOA*'s search nodes
```

```
{
 int cost; // Costo acumulado
 int stops; // Número de paradas
 struct snode* prev; // Nodo anterior en la ruta
 // Otros campos necesarios
 int state;
 unsigned g1;
 unsigned g2;
 double key;

 unsigned long heapindex;
 snode *searchtree;
};


// Variables globales
Nodo *nodosUtilizados = NULL;  // Arreglo din□mico para guardar nodos utilizados
int totalNodosUtilizados = 0; // Contador de nodos utilizados

gnode* graph_node;
unsigned num_gnodes;
unsigned adjacent_table[MAXNODES][MAXNEIGH];
unsigned pred_adjacent_table[MAXNODES][MAXNEIGH];
unsigned goal, start;
gnode* start_state;
gnode* goal_state;
snode* start_node;

unsigned long long int stat_expansions = 0;
unsigned long long int stat_generated = 0;
unsigned long long int minf_solution = LARGE;
```

```
unsigned solutions[MAX_SOLUTIONS][2];
unsigned nsolutions = 0;
unsigned stat_pruned = 0;
unsigned stat_created = 0;


//********************************************** Binary Heap Data Structures
*****************************************************

// -------------------------- Binary Heap for Dijkstra ---------------------------------------
#define HEAPSIZEDIJ 3000000
gnode* heap_dij[HEAPSIZEDIJ];
unsigned long int heapsize_dij = 0;
unsigned long int stat_percolations = 0;

// ------------------------------------------------------------
void percolatedown_dij(int hole, gnode* tmp) {
 int child;

 if (heapsize_dij != 0) {
  for (; 2 * hole <= heapsize_dij; hole = child) {
   child = 2 * hole;
   if (child != heapsize_dij && heap_dij[child + 1]->key < heap_dij[child]->key)
    ++child;
   if (heap_dij[child]->key < tmp->key) {
    heap_dij[hole] = heap_dij[child];
    heap_dij[hole]->heapindex = hole;
    ++stat_percolations;
   }
   else
```

```
      break;
    } // end for
    heap_dij[hole] = tmp;
    heap_dij[hole]->heapindex = hole;
  }
}
/* ------------------------------------------------------------- */
void percolateup_dij(int hole, gnode* tmp) {
  if (heapsize_dij != 0) {
    for (; hole > 1 && tmp->key < heap_dij[hole / 2]->key; hole /= 2) {
      heap_dij[hole] = heap_dij[hole / 2];
      heap_dij[hole]->heapindex = hole;
      ++stat_percolations;
    }
    heap_dij[hole] = tmp;
    heap_dij[hole]->heapindex = hole;
  }
}
/* ------------------------------------------------------------- */
void percolateupordown_dij(int hole, gnode* tmp) {
  if (heapsize_dij != 0) {
    if (hole > 1 && heap_dij[hole / 2]->key > tmp->key)
      percolateup_dij(hole, tmp);
    else
      percolatedown_dij(hole, tmp);
  }
}
/* ------------------------------------------------------------- */
void insertheap_dij(gnode* thiscell) {

  if (thiscell->heapindex == 0)
```

```
    percolateup_dij(++heapsize_dij, thiscell);
  else
    percolateupordown_dij(thiscell->heapindex, heap_dij[thiscell->heapindex]);
}
/* ------------------------------------------------------------ */
void deleteheap_dij(gnode* thiscell) {
  if (thiscell->heapindex != 0) {
    percolateupordown_dij(thiscell->heapindex, heap_dij[heapsize_dij--]);
    thiscell->heapindex = 0;
  }
}
/* ------------------------------------------------------------ */
gnode* topheap_dij() {
  if (heapsize_dij == 0)
    return NULL;
  return heap_dij[1];
}
/* ------------------------------------------------------------ */
void emptyheap_dij() {
  int i;

  for (i = 1; i <= heapsize_dij; ++i)
    heap_dij[i]->heapindex = 0;
  heapsize_dij = 0;
}

/* ------------------------------------------------------------ */
gnode* popheap_dij() {
  gnode* thiscell;

  if (heapsize_dij == 0)
```

```
  return NULL;
 thiscell = heap_dij[1];
 thiscell->heapindex = 0;
 percolatedown_dij(1, heap_dij[heapsize_dij--]);
 return thiscell;
}

int sizeheap_dij() {
 return heapsize_dij;
}

gnode* posheap_dij(int i) {
 return heap_dij[i];
}

// -------------------------   Binary Heap for BOA*  ---------------------------------------
#define HEAPSIZE 40000000
snode* heap[HEAPSIZE];
unsigned long int heapsize = 0;

// ------------------------------------------------------------
void percolatedown(int hole, snode* tmp) {
 int child;

 if (heapsize != 0) {
  for (; 2 * hole <= heapsize; hole = child) {
   child = 2 * hole;
   if (child != heapsize && heap[child + 1]->key < heap[child]->key)
    ++child;
   if (heap[child]->key < tmp->key) {
    heap[hole] = heap[child];
```

```
        heap[hole]->heapindex = hole;
        ++stat_percolations;
      }
      else
        break;
    } // end for
    heap[hole] = tmp;
    heap[hole]->heapindex = hole;
  }
}
/* ------------------------------------------------------------- */
void percolateup(int hole, snode* tmp) {
  if (heapsize != 0) {
    for (; hole > 1 && tmp->key < heap[hole / 2]->key; hole /= 2) {
      heap[hole] = heap[hole / 2];
      heap[hole]->heapindex = hole;
      ++stat_percolations;
    }
    heap[hole] = tmp;
    heap[hole]->heapindex = hole;
  }
}
/* ------------------------------------------------------------- */
void percolateupordown(int hole, snode* tmp) {
  if (heapsize != 0) {
    if (hole > 1 && heap[hole / 2]->key > tmp->key)
      percolateup(hole, tmp);
    else
      percolatedown(hole, tmp);
  }
}
```

```
/* ------------------------------------------------------------ */
void insertheap(snode* thiscell) {
 if (thiscell->heapindex == 0)
  percolateup(++heapsize, thiscell);
 else
  percolateupordown(thiscell->heapindex, heap[thiscell->heapindex]);
}
/* ------------------------------------------------------------ */
void deleteheap(snode* thiscell) {
 if (thiscell->heapindex != 0) {
  percolateupordown(thiscell->heapindex, heap[heapsize--]);
  thiscell->heapindex = 0;
 }
}
/* ------------------------------------------------------------ */
snode* topheap() {
 if (heapsize == 0)
  return NULL;
 return heap[1];
}
/* ------------------------------------------------------------ */
void emptyheap() {
 int i;

 for (i = 1; i <= heapsize; ++i)
  heap[i]->heapindex = 0;
 heapsize = 0;
}

/* ------------------------------------------------------------ */
snode* popheap() {
```

```
  snode* thiscell;

  if (heapsize == 0)
   return NULL;
  thiscell = heap[1];
  thiscell->heapindex = 0;
  percolatedown(1, heap[heapsize--]);
  return thiscell;
}

int sizeheap() {
  return heapsize;
}

long int opensize() {
  return heapsize_dij;
}
snode* posheap(int i) {
  return heap[i];
}
// --------------------------  Binary Heap end ------------------------------------------


//********************************************** Reading the file
*****************************************************

void read_adjacent_table(const char* filename) {
        FILE* f;
        int i, ori, dest, dist, t;
```

```
        f = fopen(filename, "r");
        int num_arcs = 0;
        if (f == NULL)  {
                printf("Cannot open file %s.\n", filename);
                exit(1);
        }
        fscanf(f, "%d %d", &num_gnodes, &num_arcs);
        fscanf(f, "\n");
//      printf("%d %d", num_gnodes, num_arcs);
        for (i = 0; i < num_gnodes; i++)
                adjacent_table[i][0] = 0;

        for (i = 0; i < num_arcs; i++) {
                fscanf(f, "%d %d %d %d\n", &ori, &dest, &dist, &t);
        //      printf("%d %d %d %d\n", ori, dest, dist, t);
                adjacent_table[ori - 1][0]++;
                adjacent_table[ori - 1][adjacent_table[ori - 1][0] * 3 - 2] = dest - 1;
                adjacent_table[ori - 1][adjacent_table[ori - 1][0] * 3 - 1] = dist;
                adjacent_table[ori - 1][adjacent_table[ori - 1][0] * 3] = t;

                pred_adjacent_table[dest - 1][0]++;
                pred_adjacent_table[dest - 1][pred_adjacent_table[dest - 1][0] * 3 - 2] = ori
- 1;
                pred_adjacent_table[dest - 1][pred_adjacent_table[dest - 1][0] * 3 - 1] = dist;
                pred_adjacent_table[dest - 1][pred_adjacent_table[dest - 1][0] * 3] = t;
        }
        fclose(f);
}

void new_graph() {
        int y;
```

```
        if (graph_node == NULL) {
                graph_node = (gnode*) calloc(num_gnodes, sizeof(gnode));
                for (y = 0; y < num_gnodes; ++y)                {
                        graph_node[y].id = y;
                        graph_node[y].gmin = LARGE;
                        graph_node[y].h1 = LARGE;
                        graph_node[y].h2 = LARGE;
                }
        }
}


//************************************************                BOA*
**********************************************************

void initialize_parameters() {
  start_state = &graph_node[start];
  goal_state = &graph_node[goal];
  stat_percolations = 0;
}

int backward_dijkstra(int dim) {
  int i;
        for (i = 0; i < num_gnodes; ++i)
    graph_node[i].key = LARGE;
  emptyheap_dij();
  goal_state->key = 0;
  insertheap_dij(goal_state);

  while (topheap_dij() != NULL) {
    gnode* n;
```

```
        gnode* pred;
        short d;
        n = popheap_dij();
        if (dim == 1)
          n->h1 = n->key;
        else
          n->h2 = n->key;
        ++stat_expansions;
        for (d = 1; d < pred_adjacent_table[n->id][0] * 3; d += 3) {
          pred = &graph_node[pred_adjacent_table[n->id][d]];
          int new_weight = n->key + pred_adjacent_table[n->id][d + dim];
          if (pred->key > new_weight) {
            pred->key = new_weight;
            insertheap_dij(pred);
          }
        }
      }
      return 1;
    }

    snode* new_node() {
      snode* state = (snode*)malloc(sizeof(snode));
      state->heapindex = 0;
      return state;
    }

    int boastar() {
      FILE* f = fopen("Txt/NodosUtilizados2.txt", "a");
      if (f == NULL) {
        perror("Error al abrir el archivo para guardar los nodos utilizados");
        exit(1);
```

```
    }

    snode* recycled_nodes[MAX_RECYCLE];
    int next_recycled = 0;
    nsolutions = 0;
    stat_pruned = 0;
    emptyheap();

    start_node = new_node();
    ++stat_created;
    start_node->state = start;
    start_node->g1 = 0;
    start_node->g2 = 0;
    start_node->key = 0;
    start_node->searchtree = NULL;
    insertheap(start_node);

    stat_expansions = 0;
    while (topheap() != NULL) {
      snode* n = popheap(); // Best node in open
      short d;

      if (n->g2 >= graph_node[n->state].gmin || n->g2 + graph_node[n->state].h2 >=
minf_solution) {
        stat_pruned++;
        if (next_recycled < MAX_RECYCLE) {
          recycled_nodes[next_recycled++] = n;
        }
        continue;
      }
```

```
    graph_node[n->state].gmin = n->g2;

    // Guardar el nodo expandido
    fprintf(f, "Nodo expandido: %d (g1: %d, g2: %d)\n", n->state, n->g1, n->g2);

    if (n->state == goal) {
      printf("GOAL [%d,%d] nsolutions:%d expanded:%llu generated:%llu heapsize:%d pruned:%d\n",
          n->g1, n->g2, nsolutions, stat_expansions, stat_generated, sizeheap(), stat_pruned);
      solutions[nsolutions][0] = n->g1;
      solutions[nsolutions][1] = n->g2;
      nsolutions++;
      if (nsolutions > MAX_SOLUTIONS) {
        printf("Maximum number of solutions reached, increase MAX_SOLUTIONS!\n");
        fclose(f);
        exit(1);
      }
      if (minf_solution > n->g2)
        minf_solution = n->g2;
      continue;
    }

    ++stat_expansions;

    for (d = 1; d < adjacent_table[n->state][0] * 3; d += 3) {
      snode* succ;
      double newk1, newk2, newkey;
      unsigned nsucc = adjacent_table[n->state][d];
      unsigned cost1 = adjacent_table[n->state][d + 1];
      unsigned cost2 = adjacent_table[n->state][d + 2];
```

```
        unsigned newg1 = n->g1 + cost1;
        unsigned newg2 = n->g2 + cost2;
        unsigned h1 = graph_node[nsucc].h1;
        unsigned h2 = graph_node[nsucc].h2;

        if (newg2 >= graph_node[nsucc].gmin || newg2 + h2 >= minf_solution)
          continue;

        newk1 = newg1 + h1;
        newk2 = newg2 + h2;

        if (next_recycled > 0) { // To reuse pruned nodes in memory
          succ = recycled_nodes[--next_recycled];
        } else {
          succ = new_node();
          ++stat_created;
        }

        succ->state = nsucc;
        stat_generated++;

        newkey = newk1 * (double)BASE + newk2;
        succ->searchtree = n;
        succ->g1 = newg1;
        succ->g2 = newg2;
        succ->key = newkey;
        insertheap(succ);
      }
    }
```

```
    fclose(f);
    return nsolutions > 0;
}



/* ------------------------------------------------------------------------*/
void call_boastar() {
    float runtime;
    struct timeval tstart, tend;
    unsigned long long min_cost;
    unsigned long long min_time;

    initialize_parameters();

    gettimeofday(&tstart, NULL);

    //Dijkstra h1
    if (backward_dijkstra(1))
        min_cost = start_state->h1;

    //Dijkstra h2
    if (backward_dijkstra(2))
        min_time = start_state->h2;


            //BOA*
    boastar();
```

```
    gettimeofday(&tend, NULL);
    runtime = 1.0 * (tend.tv_sec - tstart.tv_sec) + 1.0 * (tend.tv_usec - tstart.tv_usec) / 1000000.0;
    // printf("nsolutions:%d Runtime(ms):%f Generated: %llu statexpanded1:%llu\n", nsolutions, time_astar_first1*1000, stat_generated, stat_expansions);
    printf("%lld;%lld;%d;%f;%llu;%llu;%lu;%llu\n",
        start_state->id + 1,
        goal_state->id + 1,
        nsolutions,
        runtime * 1000,
        stat_generated,
        stat_expansions,
        stat_created,
        stat_percolations);
}

void execute_with_stops(const int stops[], int num_stops) {
    unsigned long long total_expansions = 0, total_generated = 0;
    float total_runtime = 0;
    unsigned total_solutions = 0;

    struct timeval tstart, tend;
    int i;

    for (i = 0; i < num_stops - 1; i++) {
        start = stops[i];     // Nodo inicial del segmento
        goal = stops[i + 1];   // Nodo final del segmento
        printf("Processing segment: %d -> %d\n", start, goal);

        start_state = &graph_node[start];
        goal_state = &graph_node[goal];
```

```
    gettimeofday(&tstart, NULL);

    // Inicializar par�metros y ejecutar BOA* para este segmento
    initialize_parameters();
    if (backward_dijkstra(1) && backward_dijkstra(2)) {
      boastar();
    }

    gettimeofday(&tend, NULL);

    // Calcular tiempo de ejecuci�n para este segmento
    float runtime = 1.0 * (tend.tv_sec - tstart.tv_sec) +
            1.0 * (tend.tv_usec - tstart.tv_usec) / 1000000.0;


    // Acumular m�tricas globales
    total_runtime += runtime;
    total_expansions += stat_expansions;
    total_generated += stat_generated;
    total_solutions += nsolutions;

    // Mostrar resultados del segmento
    printf("Segment results: Runtime %.2f ms, Solutions %d, Expansions %llu, Generated
%llu\n",
        runtime * 1000, nsolutions, stat_expansions, stat_generated);

  }

  // Mostrar resultados consolidados
  printf("Total Results:\n");
```

```
    printf("Runtime: %.2f ms\n", total_runtime * 1000);
    printf("Total Expansions: %llu\n", total_expansions);
    printf("Total Generated: %llu\n", total_generated);
    printf("Total Solutions: %u\n", total_solutions);
}

/*-------------------------------------------------------------------------------*/

// Implementaci�n para guardar nodos utilizados
void guardarNodosUtilizados(const char *nombreArchivo) {
    FILE *archivo = fopen(nombreArchivo, "a");
    if (archivo == NULL) {
        perror("Error al abrir el archivo de salida");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < totalNodosUtilizados; i++) {
        fprintf(archivo, "%d %d %d\n", nodosUtilizados[i].id, nodosUtilizados[i].coordX,
nodosUtilizados[i].coordY);
    }

    fclose(archivo);
    printf("Nodos utilizados guardados en %s\n", nombreArchivo);
}

// Simulaci�n de la funci�n que copia un nodo
void copiarNodo(int idNodo, Nodo *destino) {
    // Aqu� se llenar�a el nodo `destino` con los datos reales desde el grafo.
    // Simulamos con datos ficticios:
    destino->id = idNodo;
    destino->coordX = idNodo * 10;  // Supongamos coordenada X
```

```
    destino->coordY = idNodo * 20;  // Supongamos coordenada Y
}

// Agregar un nodo a la lista de nodos utilizados
void agregarNodoUtilizado(int idNodo) {
    if (totalNodosUtilizados >= 1000) {
        // Redimensionar din□micamente si es necesario
        nodosUtilizados = realloc(nodosUtilizados, (totalNodosUtilizados + 500) *
sizeof(Nodo));
        if (nodosUtilizados == NULL) {
            perror("Error al redimensionar el array de nodos utilizados");
            exit(EXIT_FAILURE);
        }
    }

    copiarNodo(idNodo, &nodosUtilizados[totalNodosUtilizados]);
    totalNodosUtilizados++;
}

// Implementaci□n simulada de `execute_with_stops`



int main(int argc, char *argv[]) {
    // Verificar que se han pasado suficientes argumentos
    if (argc < 2) {
        printf("Se necesitan al menos un argumento para las paradas.\n");
        return 1;  // Salir si no se pasa ning□n argumento
    }

    // Convertir los argumentos a enteros y guardarlos en un array
```

```
    int num_stops = argc - 1;
    int stops[num_stops];

    for (int i = 1; i < argc; i++) {
      stops[i - 1] = atoi(argv[i]);
    }

    // Leer el grafo desde el archivo
    read_adjacent_table("Txt/filtered_NY-road-d.txt");
    new_graph();

    // Reservar memoria inicial para nodos utilizados
    nodosUtilizados = malloc(1000 * sizeof(Nodo));  // Tama◆o inicial arbitrario
    if (nodosUtilizados == NULL) {
      perror("Error al reservar memoria para nodos utilizados");
      return 1;
    }

    // Ejecutar BOA* para cada segmento entre paradas
    execute_with_stops(stops, num_stops);

    // Guardar nodos utilizados en un archivo
    guardarNodosUtilizados("Txt/NodosUtilizados2.txt");

    // Liberar memoria
    free(nodosUtilizados);

    return 0;
}
```

ListaParadas.py:

```
import subprocess



# Abrir el archivo de salida en modo de escritura (creará el archivo si no existe)
with open('Txt/output.txt', 'w') as archivo_salida:
  # Leer el archivo línea por línea
  with open('Txt/NY-queries-2p.txt', 'r') as archivo:
    #instancia_id = 1
    for linea in archivo:
      # Eliminar saltos de línea y espacios extra
      linea = linea.strip()
      # Dividir la línea en una lista de números
      stops = linea.split()

      #archivo_salida.write(f'Instancia {instancia_id}\n')
      # Ejecutar el programa C y redirigir la salida al archivo
      result = subprocess.run(['C/RetoIA.exe'] + stops, stdout=archivo_salida, stderr=archivo_salida)

      #instancia_id += 1
```

ListaParadasFinal.py:

```
import subprocess



# Abrir el archivo de salida en modo de escritura (creará el archivo si no existe)
with open('Txt/output2.txt', 'w') as archivo_salida:
```

```
    # Leer el archivo línea por línea
    with open('Txt/NY-queries-2p.txt', 'r') as archivo:
        for linea in archivo:
            # Eliminar saltos de línea y espacios extra
            linea = linea.strip()
            # Dividir la línea en una lista de números
            stops = linea.split()
            # Ejecutar el programa C y redirigir la salida al archivo
            result = subprocess.run(['C/RetoIA - Final.exe'] + stops, stdout=archivo_salida, stderr=archivo_salida)
```

RelacionarTXT.py:

```
# Archivos de entrada y salida
nodos_utilizados_file = "Txt/NodosUtilizados.txt"
road_file = "Txt/NY-road-d.txt"
output_file = "Txt/filtered_NY-road-d.txt"



# Extraer nodos únicos de nodosutilizados.txt
nodos_utilizados = set()

with open(nodos_utilizados_file, "r") as file:
    for line in file:
        if line.strip() and "Nodo expandido:" in line:
            # Extraer el ID del nodo expandido
            parts = line.split(":")
            nodo = parts[1].strip().split()[0]
            nodos_utilizados.add(nodo)
```

```
# Filtrar conexiones del archivo NY-road-d.txt
with open(road_file, "r") as road, open(output_file, "w") as output:
    output.write(f"264346 730100\n")
    for line in road:
        parts = line.strip().split()
        if len(parts) == 4:  # Validar formato correcto
            nodo_origen, nodo_destino = parts[0], parts[1]
            if nodo_origen in nodos_utilizados or nodo_destino in nodos_utilizados:
                output.write(line)

print(f"Archivo filtrado generado: {output_file}")
```

ScriptIA.py:

```
import subprocess
import os
import time  # Importar módulo time para medir el tiempo

# Definir la carpeta que contiene los scripts Python
scripts_folder = "Python"

# Lista de los archivos Python que deseas ejecutar en orden
scripts_to_run = [
    "ListaParadas.py",
    "RelacionarTXT.py",
    "ListaParadasFinal.py"
]

# Función para ejecutar un script Python
def run_script(script_name):
    script_path = os.path.join(scripts_folder, script_name)
    try:
```

```
        print(f"Ejecutando {script_name}...")
        # Usar 'python' en lugar de 'python3' para Windows
        result = subprocess.run(['python', script_path], check=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
        print(f"Salida de {script_name}: {result.stdout}")
    except subprocess.CalledProcessError as e:
        # Manejar el error de salida y decodificar stderr de forma segura
        print(f"Error al ejecutar {script_name}: {e.stderr.encode('utf-8', 'ignore').decode('utf-8')}")
        return False
    return True

# Medir el tiempo de ejecución total
start_time = time.time()  # Registrar el tiempo al inicio

# Ejecutar cada script de la lista
for script in scripts_to_run:
    if not run_script(script):
        print(f"Abortando ejecución debido a error en {script}.")
        break
else:
    print("Todos los scripts se ejecutaron correctamente.")

# Calcular el tiempo total de ejecución
end_time = time.time()  # Registrar el tiempo al final
total_time = end_time - start_time  # Diferencia entre los tiempos

print(f"Tiempo total de ejecución: {total_time:.2f} segundos.")
```